Science of Computer Programming 74 (2009) 296–313

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

A study of potential parallelism among traces in Java programs

Borys J. Bradel *, Tarek S. Abdelrahman

Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, Toronto, Ontario, Canada M5S 3G4

ARTICLE INFO

Article history:
Available online 7 February 2009

Keywords:
Traces
Parallelism
Code optimization
Java
Benchmark characterization

ABSTRACT

The exploitation of parallelism among traces, i.e. hot paths of execution in programs, is a novel approach to the automatic parallelization of Java programs and it has many advantages. However, to date, the extent to which parallelism exists among traces in programs has not been made clear. The goal of this study is to measure the amount of trace-level parallelism in several Java programs. We extend the Jupiter Java Virtual Machine with a simulator that models an abstract parallel system. We use this simulator to measure trace-level parallelism. We further use it to examine the effects of the number of processors, trace window size, and communication type and cost on performance. We also analyze the dependence characteristics of the benchmarks and see how they relate to parallelism. The results indicate that enough trace-level parallelism exists for a modest number of processors. Thus, we conclude that trace-based parallelization is a potentially viable approach to improve the performance of Java programs.

1. Introduction

Computers that contain two or more processors are becoming more popular, and companies are either putting multiple processors on a single chip, or are planning to do so [20]. Hardware that contains multiple processors, however, requires software that can execute in parallel on multiple processors. Such software is time consuming and difficult to write. Furthermore, there are many programs that have already been written to execute sequentially on a single processor. Automatically parallelizing sequential software can both ease the development of new software that effectively utilizes multiple processors and ensure that applications that have already been written will benefit from the increasing capabilities of hardware.

Automatically parallelizing software, however, is an open research problem. Traditionally, only scientific applications with regular array accesses have been successfully parallelized automatically [27]. Other types of programs have memory access patterns that are too complicated to analyze. We examine a novel approach to automatic parallelization: use traces, which are sequences of frequently executed instructions, as the unit of parallel work. We hope that using traces while incorporating the existing approaches will yield an effective automatic parallelization solution.

However, before embarking on this approach, it is beneficial to determine the extent to which parallelism exists among traces. Thus, in this paper we explore the feasibility of using traces by examining the amount of trace-level parallelism that exists in Java programs. More specifically, we extend an existing Java Virtual Machine (JVM), Jupiter [16], that executes programs sequentially by allowing it to simulate the parallel execution of traces. The simulated execution is used to compute the amount of parallelism. We consider the effects of factors such as the number of processors, interprocessor communication cost, and communication type on parallel performance. Finally, we characterize the behaviour of traces based on where they read data from. The results indicate that most of the benchmarks have enough trace-level parallelism to result in good performance on systems with up to four processors, with some benchmarks being able to scale to eight processors. The results also indicate that for the remaining benchmarks the main source of such poor performance is the lack

* Corresponding author.
E-mail addresses: bradel@eecg.toronto.edu (B.J. Bradel), tsa@eecg.toronto.edu (T.S. Abdelrahman).

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2009.01.004

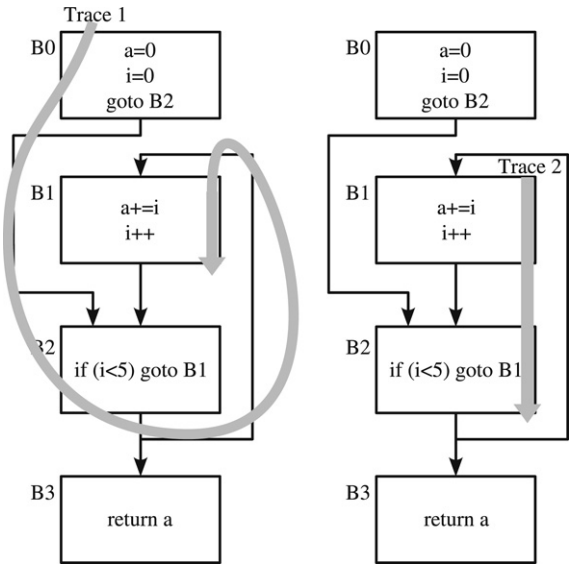

Fig. 1. Traces mapped onto a control flow graph.

of parallelism as opposed to high interprocessor communication costs. Therefore, we conclude that exploiting trace-level parallelism is a viable approach to improve the performance of Java programs.

The remainder of this paper is organized as follows: Section 2 contains background material. Section 3 describes the execution model that we use. Section 4 describes the simulated parallel system. Section 5 contains some implementation details of the simulator. Section 6 contains the experimental evaluation. Section 7 gives related work. Finally, Section 8 presents concluding remarks and future work. An Appendix is included that contains the simulation methodology and timing and complete performance results.

2. Background

In this section, traces are defined, their potential benefits are discussed, and Jupiter and RedSpot are described.

2.1. Traces

A trace is a sequence of *n unique* basic blocks,[1] $(b_1, b_2, \ldots, b_n)$, such that basic blocks $b_1, b_2, \ldots, b_n$ are executed in sequential order during the execution of a program [3]. Block b_1 is called the *start* of the trace and b_n is called the *end* of the trace. The trace may contain any basic blocks of the program as long as the sequence corresponds to a path on the control flow graph. Since a trace can contain multiple basic blocks it can contain multiple points at which control flow can exit the trace. These points are referred to as *trace exits*.

An example of traces is in Fig. 1, which contains two copies of a control flow graph, each with a different valid trace. The traces are $(B0, B2, B1)$ and $(B1, B2)$. In contrast, the sequence $(B1, B2, B1)$ is not a valid trace because $B1$ appears twice.

Traces are generated by a trace collection system (TCS) that monitors a program's execution and collects traces based on this execution [4]. The TCS starts recording a trace when occurrences of certain events, such as a backward taken branch, a backward jump, and a trace exit, exceed a specific threshold. The recording stops when certain events occur, such as a backward taken branch or jump, or the start of a recorded trace. Once the traces are collected, they can be used for optimization.

2.2. Benefits of using traces

The use of traces as units of parallel work offers a number of potential benefits for automatic parallelization [8]. First, traces are based on a binary representation of a program, which for this work is Java bytecode. Therefore traces allow the automatic parallelization of programs without the need to examine source code. This is a significant advantage since in most real world scenarios the source code of the entire program is not available for examination by a compiler. Second, traces can be parts of loops, can be parts of individual methods, or can span multiple methods. Traces may also be combined to incorporate multiple loops and methods. Therefore traces subsume loop iterations and methods as units of parallel work, thus offering more flexibility than either loop iterations or methods and exhibiting both data and task level parallelism.

[1] A basic block is a maximal sequence of consecutive instructions for which execution must start with the first instruction and continue until the last instruction [7].

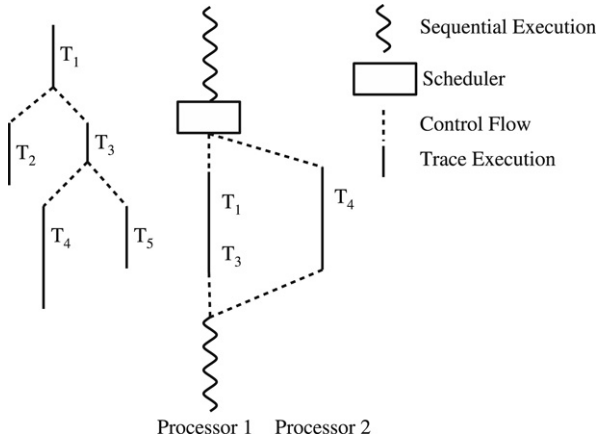

Fig. 2. Traces executing according to the proposed execution model.

Third, traces restrict the control flow that is considered for parallelization. The additional runtime information can be used to avoid spending time trying to parallelize infrequently executed sections of a program and to improve the analysis that is performed, which hopefully increases the accuracy of the analysis. Finally, traces are collected by keeping track of program execution, and are relatively simple to identify. Thus, compared to other approaches that attempt to create units of work other than loops and methods, e.g. for speculative execution [24], traces can be collected with a low overhead that will have a negligible effect on overall execution time [33].

2.3. Jupiter and RedSpot

The underlying infrastructure used to identify traces consists of Jupiter [16], an interpreter based JVM, and RedSpot [7], a trace collection system that is an extension of Jupiter. Jupiter's interpreter performs three distinct tasks for each interpreted bytecode instruction: the bytecode is executed by updating the appropriate data structures in Jupiter, the next bytecode to be executed is identified, and Jupiter's state is updated to enable the interpretation of the next bytecode. Also, a fourth task is performed for all bytecodes that represent control flow instructions: RedSpot is called to allow it to identify the basic blocks and traces executed by Jupiter based on the current and next bytecode.

RedSpot counts the number of times that certain events occur. Each event is the encountering of a specific return, trace exit, or backward taken branch or jump. When the number of times that an event is encountered reaches a certain recording threshold, recording of a trace starts. The recording of the trace stops when a backward branch or jump or the start of another trace is encountered [7].

3. Model of parallel execution

A sequential program that contains many traces can be executed in parallel by making the traces execute in different threads of execution. The following is our model for the parallel execution of a program based on traces [8]. A program begins to execute instructions, which are not on traces, sequentially. A trace collection system monitors this execution, detects traces, and saves them. When the program execution reaches one of these saved traces, the trace is assigned to execute on a new thread and the most likely successor trace is identified. This trace is then assigned to execute on a new thread, and its own most likely successor is in turn identified. The threads of the identified traces are then given to a scheduler, which assigns the threads to processors. The threads are then executed.[2] Eventually, execution will go off trace and the next instructions that need to be executed are not on traces. These instructions are executed sequentially until the next set of traces is encountered, at which point parallel execution resumes. The process repeats until the end of the program.

An example of this execution model is shown in Fig. 2. The left side of the figure contains traces T_1, T_2, T_3, T_4, and T_5. T_3 is the most likely successor of T_1, and T_4 is the most likely successor of T_3. The right side of the figure contains the execution of the program. T_1 is assigned to execute in a new thread, and T_3 is identified as its most likely successor. Then T_3 is assigned to execute in a new thread, and T_4 is identified as its most likely successor. Finally, T_4 is assigned to execute in a new thread. Once T_1, T_3, and T_4 are assigned to new threads, they are scheduled: T_1 and T_3 are scheduled to execute on one processor, and T_4 is scheduled to execute on the second processor. After the traces complete executing, sequential execution resumes.

The question that is addressed in this work is: *how much parallelism exists when applications are executed using this model.* This question is answered by simulating the execution of several applications on an abstract parallel system that satisfies the

[2] The way in which traces execute is implementation specific, and the approach used in this paper is described in Section 4.

execution model presented above. The results are not an indication of the performance of a real system. Rather, they are an indication of the amount of parallelism that exists in applications, which is what this work is concerned with. Consequently, simplifying assumptions are made that may result in more parallelism than can be realized in actual hardware systems. We believe that such assumptions are acceptable because we are measuring an upper bound on performance. Thus, this work is separate from how to efficiently exploit the parallelism that exists, which is not addressed in this paper.

4. Parallel system

The simulator keeps track of a program's Java bytecodes and the memory locations that are accessed as the bytecodes are executed by a JVM. Instructions that are not on a trace are executed sequentially. Instructions on traces are executed in parallel, and scheduled in a way that enforces the data dependences that exist among the traces.

The simulation measures the total number of cycles that are required to execute an application in parallel relative to executing the application sequentially. These summary scheduling results are used to compute an *abstract speedup* of the system. This metric is used to measure the trace-level parallelism that exists in Java programs.

Although the abstract speedup is based on a single execution of a program with a specific input set, the evaluation in Section 6 is based on benchmarks with input sets that make the benchmark execute in a representative fashion. Therefore the experimental evaluation reflects an abstract speedup that is representative of the behaviour of the benchmarks with different inputs and thus the evaluation is sound.

The scheduler takes communication of data into account. When reads and the writes that these reads depend on are on different processors, the scheduler ensures that each read is executed after the data is transmitted to the read's processor. Therefore when such reads are executed, the correct data is always available for them to use, and no explicit synchronization primitives need to be inserted on the traces to ensure that the data being read is correct.

The maximum number of traces that can be scheduled together is referred to as the *trace window size*. At the end of each trace window, the scheduler is called to schedule the traces in that window on the abstract system. Scheduling is performed with precise knowledge of which traces execute in each window. This information is obtained from the JVM. Therefore control flow is predicted correctly within each window. However, no information is kept between traces in different windows and such traces cannot be scheduled to execute in parallel. Thus, the size of the trace window limits parallelism.

There are advantages and disadvantages of different window sizes. The advantage of a large window size is that the maximum possible parallelism can be achieved. The disadvantage is that the simulated performance does not reflect the behaviour of a real system. The system assumes that all the traces in each window are going to execute. This assumption is valid for the simulator. However, in a real system, prediction needs to identify which traces are going to execute. Since the prediction cannot always be correct, the more traces need to be predicted, the lower the probability of correctly predicting all of them, and the higher the overhead cost of dealing with the incorrect predictions.[3] Therefore performance on a real system would degrade as the number of traces increases. Thus, a large window size causes the simulation to diverge from the behaviour of a real system. The advantage of a smaller window size is that it makes the simulation more realistic and the disadvantage is that less parallelism can be identified. A good choice for a window size is one that allows a reasonable amount of parallelism to be identified and contains few enough traces that a real system could predict the execution of the traces with a high level of probability.

The remainder of this section contains a description of our parallel system model. Although the model is too simplistic to precisely capture a realistic parallel system, the model captures sufficient details to give an indication of the parallelism that exists among traces.

4.1. Processor model

The abstract system contains a fixed number of processors. Each processor executes instructions sequentially, in order, and one at a time. All data is in memory and there are no registers. The memory consists of both the heap and the stack. This model corresponds to the way that Java bytecodes access data without using registers. The number of cycles that a processor takes to execute an instruction is equal to the number of words the instruction writes to memory. Most instructions will take one cycle to execute, which is the ideal behaviour of a pipelined processor that can complete one instruction per cycle. Since control flow instructions do not write data, they take zero cycles to execute. Instructions that write multiple words either deal with floating point values or are sequences of simple instructions that are combined for conciseness. Thus, such instructions should take longer to execute, and this behaviour is captured by making the number of cycles that an instruction executes be equal to the number of words that the instruction writes.

The memory is shared between all processors and data can be read directly from memory without incurring additional delays. However, if data is read on one processor when it was written before on another processor, then the read must wait until the data is transmitted between the processors. This behaviour is similar to having the working set fit in a cache and having to miss in the cache when another processor invalidates the data.

[3] Studying the effects of different prediction heuristics and misprediction costs is left as future work.

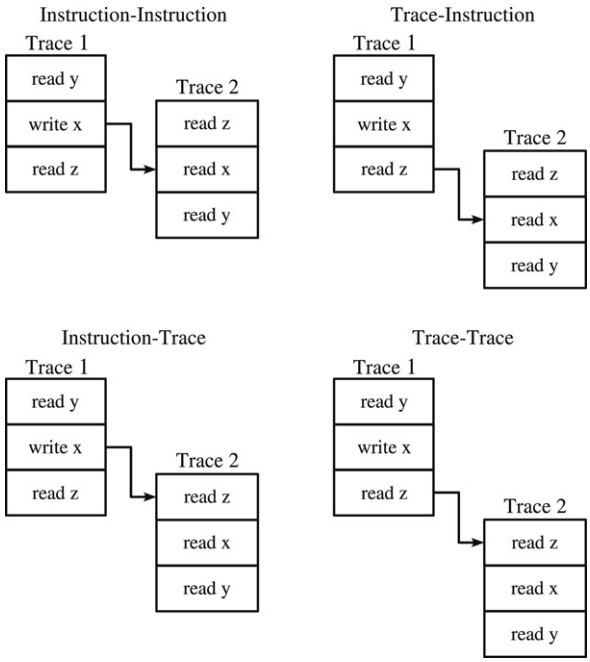

Fig. 3. Different types of dependences.

4.2. Communication model

Communication between processors occurs when an instruction on one processor writes a value that is read by an instruction on another processor. Two factors affect when data that an instruction depends on is ready for that instruction's use.

The first is whether the data is written and read on the same processor or not. If the read and write are on different processors, then communication cost is incurred. The cost has two components: a fixed cost for all communication between a pair of traces on separate processors, and a cost that is proportional to the number of words being transmitted. The *fixed communication cost* is defined as the number of cycles that need to elapse between two communicating traces on separate processors. The *variable communication cost* is defined as the number of cycles that need to elapse for each word that is transmitted between the two traces. The communication cost between any two traces is the fixed communication cost plus the number of reads times the variable communication cost.

The second factor is whether data is shared between the beginnings and ends of traces or instructions. Four possible combinations exist:

- the data is ready at the end of the write instruction and can be read immediately;
- the data is ready at the end of the trace that the write is on, and the data needs to be read at the beginning of the trace that the read is on;
- the data is ready at the end of the write instruction, and the data needs to be read at the beginning of the trace that the read is on; and
- the data is ready at the end of the trace that the write is on and can be read immediately.

Each of these is referred to as *instruction–instruction*, *trace–trace*, *instruction–trace*, and *trace–instruction communication*, respectively. The different types of communication impact the amount of parallelism because they control the amount of overlap between communication and execution. These combinations are shown in Fig. 3, which contains two traces, one writing `x` and one reading `x`.

Preserving all the dependences between reads and writes would under-report the amount of parallelism available. The reason is that some instructions modify data in an inherently sequential manner, even though this data can be precomputed. An example of such behaviour is the updating of induction variables. Therefore, if other instructions depend on the data written by such special instructions, this dependence must be removed. All instructions that perform a read and a write from the same location are classified as *induction instructions* (e.g. $\mathtt{i} = \mathtt{i} + 1$), and assume that any subsequent instruction that reads the produced data will be dependent on the instruction that initialized the data before the induction occurred. Thus, the dependences caused by these instructions are removed. This classification identifies the primary induction variables in all the benchmarks that are used in the experimental evaluation. Although removing more dependences may lead to more parallelism, taking into account all possible ways of removing dependences is not feasible for this study. Removing more

dependences is left as future work. Removing dependences will only improve the amount of parallelism available, thus enabling more parallelism, and making traces a more viable parallelization approach.

4.3. Dependence characterization

The dependences between traces can be classified based on the relative location of the reads and writes that cause dependences. A trace can read data from three different sources.[4] The first is data produced by the trace itself. The second is data produced by another trace scheduled on the same processor. Finally, the third source is data produced by another trace that executed on a different processor. Reads that use these three different sources respectively are referred to as *on-trace reads*, *on-processor reads*, and *off-processor reads*. Only the last type incurs an overhead of communication between processors. This classification can be performed only after scheduling is performed. It is only after scheduling that on-processor and off-processor reads can be distinguished.

Traces are classified according to three groups: those that contain some off-processor reads, those that contain some on-processor reads but no off-processor reads, and those that only contain on-trace reads. The traces in these groups are referred to as *remote*, *local*, and *isolated* traces respectively. Examining each group of traces separately allows a more detailed analysis of the sources of parallel overhead.

The off-processor reads are further subdivided into *critical reads* and *non-critical reads*. A critical read is an off-processor read on a trace that has been scheduled close in time to the trace that generates the data being read. Close in time is defined as being within the communication cost between processors. These reads are critical because any delay in communication will hinder performance. The non-critical reads are off-processor reads on a trace that has been scheduled far apart in time to the trace that generates the data being read. These reads have less of an impact on performance because even though they incur communication overhead, this overhead can be hidden by executing useful work.

The above classification allows us to reason about how dependences influence performance. An application that has many isolated traces should have a large amount of parallelism. Similarly, an application with many remote traces indicates that although data dependences exist, it is possible to schedule traces on multiple processors. Nonetheless, the presence of these dependences may affect performance due to communication overhead, especially when a large number of critical reads exists. In contrast, an application that has a majority of local traces will probably have little parallelism. The reason is that local traces are traces that had to be scheduled on the same processor with the traces that they depend on, thus causing serialization. Conversely, a small number of local traces and reads on local traces should indicate that a large amount of parallelism exists. Therefore, there are two likely sources of performance loss. The first is communication overhead incurred by critical reads. The second is lack of parallelism because of serialization, which is indicated by large numbers of local traces. The impact of these sources is examined in Section 6.4.

5. Implementation

We have created a simulator that extends Jupiter [16] and RedSpot [7] to measure trace-level parallelism. This simulator is called PIE, which stands for Parallelism Identification Engine. PIE receives information from Jupiter and RedSpot, generates scheduling information one trace window at a time, and aggregates the results to produce a measure of parallelism.

PIE is informed about the instructions that execute, the memory that these instructions read and write, and the traces that these instructions are on. This information is used to keep track of dependences. The dependences are between each memory location and the write that modifies that location at a given moment in time and between reads and writes. The dependences are used to schedule the traces on the abstract parallel system.

PIE is given this information in the form of events from Jupiter and RedSpot. The events are generated as the instructions are executed sequentially by Jupiter. Thus the traces are generated based on a sequential execution of the program and there is no scheduler which could affect the traces. At the beginning of each instruction, an event is sent that states which instruction executes and what trace, if any, it is on. When a memory location is read or written, an event is sent that states which memory location was accessed and whether it was read or written. Since Java bytecodes access all data through either the stack or the heap, all data has a memory location associated with it. PIE stores these events in a queue.

The queue of events is processed when PIE is informed about the execution of control flow instructions. A queue is used because the trace that an instruction is on is not always known when that instruction is executed. The reason for this behaviour is that RedSpot processes information only at control flow instructions, and uses this information to determine not only which trace the next instructions will be on, but in certain cases, which trace the instructions before the control flow instruction are on. Therefore, PIE can only process instructions in the event queue once a control flow instruction is in the queue ahead of them. When PIE is sent an event that corresponds to a control flow instruction, it processes the events in the queue and correctly assigns each instruction to an appropriate trace.

PIE uses a hash table to associate each memory location with a write and a graph to represent dependent traces. When a write is processed, the memory location being written is associated with the write and its trace. When a read is processed,

[4] Reads of constants are ignored since these values can be known ahead of time and do not cause dependences.

the write and its trace which are associated with the read location are identified. If the write and read are on the same trace or the write wrote a constant or an induction variable, then the read is assumed to be independent of the write. The reason is that the written values can be identified ahead of time. Furthermore, the number of reads that are on the same trace as writes is recorded. For all other writes, a dependence is created between the trace that performs the write and the trace that performs the read. Anti and output dependences are not an issue because the data is written as the instructions are executed by Jupiter, and is therefore written in the order of the sequential execution of the instructions. Thus no aliasing occurs.

Once enough traces execute, the dependence graph is given to a scheduler. PIE uses the modified critical-path (MCP) algorithm, which is a list scheduling algorithm that takes communication costs into consideration [32]. The algorithm calculates the schedule of a dependence graph of traces.

Each node on the graph is a trace and its weight is the number of writes on the trace, which are assumed to execute sequentially and in order. The traces are scheduled in a way that ensures that all reads occur far enough ahead of writes and therefore no extra stalls will occur.

Each edge on the graph corresponds to a dependence between two traces. The weight of the edge is computed based on the number of writes and the type of communication that is used (e.g. trace–trace or instruction–instruction). If the type of communication involves instructions, then the read–write pair that causes the largest overhead for instruction–instruction communication is used. The weight is the sum of three values: a fixed communication cost, a communication cost proportional to the number of read–write pairs between the traces, and the number of cycles that need to exist between the start and end of the trace based on the communication type.

The schedule generated for the graph indicates the speedup obtained by using multiple processors, and the aggregate result is presented in Sections 6.1–6.3.

After scheduling, the dependence graph is traversed. Each dependence is categorized based on the processor that the traces that perform the read and write are on and how many cycles elapse between the traces. This information is then aggregated and combined with the information about how many reads are on the same traces as the writes that they depend on. The aggregated information is presented in Section 6.4, where it is used to analyze the impact that dependences have on performance.

6. Experimental evaluation

The simulator, PIE, is an extension of Jupiter [16] with RedSpot [7] and uses MCP [31,32] for scheduling. RedSpot uses a recording threshold of 42, which has been shown in prior work to yield traces with good characteristics [7]. GCC 3.1.1 and the Blackdown JDK 1.4.2 are used for compilation. The experiments are run on a Socket 954 Athlon 64 3000+ processor with 512 MB of RAM with Ubuntu Linux 6.10.

The amount of available parallelism is measured for sequential applications from the Java Grande benchmark suite [23], the SPECjvm98 benchmark suite [29], and the Jolden benchmark suite [12,21]. The Java Grande applications are *moldyn*, *raytracer*, *euler*, *montecarlo*, and *search*. The SPECjvm98 applications are *compress*, *jess*, *db*, *javac*, *mpegaudio*, and *jack*. The Jolden applications are *bh*, *bisort*, *em3d*, *health*, *mst*, *perimeter*, *power*, *treeadd*, *tsp*, and *voronoi*.

The benchmark suites represent three different types of applications. Java Grande contains scientific floating point array-based programs. SPECjvm98 contains mostly general purpose integer programs with some array based applications. Finally, Jolden contains programs that extensively use dynamic data structures such as linked lists and trees. Thus, the selection of benchmark suites represents a good cross section of Java applications.

Two of the benchmarks are slightly modified to avoid limitations of Jupiter. *Jess* has been modified to make its inner classes explicitly stated[5] and *voronoi* has been modified to avoid a division by zero[6] in the `createPoints` method of the `Vertex` class. Jupiter would fail to execute these benchmarks without the changes.

The smaller of two input data sets is used for the Java Grande suite. Using the larger input data sets results in similar simulated performance and therefore showing graphs for the larger set would be redundant. The default (large) input data set is used for the benchmarks in the SPECjvm98 suite. The Jolden benchmarks must have their input data set sizes set individually. The sizes are shown in Table 1.

The first 2^{27} instructions are executed without collecting statistics and the subsequent 2^{25} instructions are executed while statistics are collected. PIE exhibits excessive memory use for two benchmarks: *mst* and *treeadd*. This behaviour limits the size of the inputs that can be used. The consequence is that less than 2^{27} instructions are executed by these benchmarks. Therefore instead of 2^{27} instructions, 2^{26} instructions are executed for *mst* and 2^{24} instructions for *treeadd* without collecting statistics. The number of instructions executed when statistics are collected does not change. These settings allow us to capture the steady state behaviour of the benchmarks.[7]

[5] Although in Java an inner class can be referred to without being defined in the source code, Jupiter requires an explicit class definition (e.g. class InnerClass { InnerClass() { return; } }) for the inner class.

[6] The division is of a double by an integer. Although other JVMs can handle this case by using floating point arithmetic, Jupiter throws an exception that causes it to crash.

[7] These numbers are derived from previous work that characterizes trace behaviour [7]. The graphs in that work indicate that trace execution is in a steady state after 2^{27} instructions and that steady state behaviour can be captured in 2^{25} instructions.

Table 1
Input set sizes for Jolden.

Benchmark	Input size
bh	1024 bodies
bisort	150,000 numbers
em3d	2000 nodes, 100 degrees
health	5 levels, 500 iterations
mst	750 nodes
perimeter	16 levels
power	Fixed size
treeadd	20 levels
tsp	17,000 cities
voronoi	20,000 points

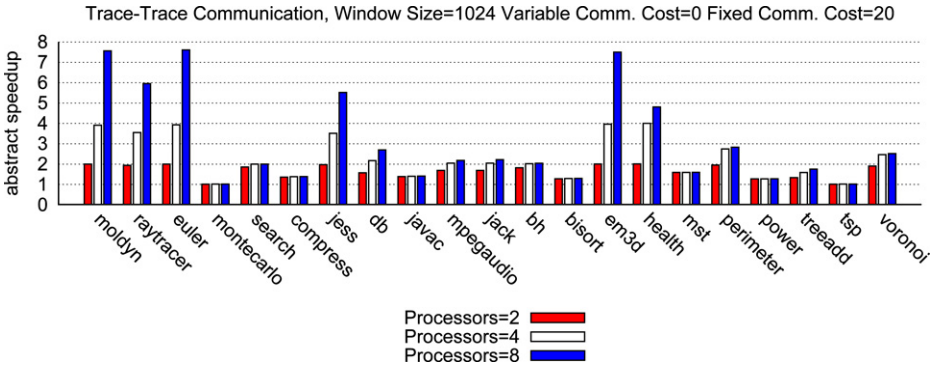

Fig. 4. Effect of different numbers of processors.

The default configuration is an abstract system with four processors, a window size of 1024 traces, and trace–trace communication with zero variable cost and a 20 cycle fixed communication cost. The effect that these parameters have on parallelism is examined by varying them and measuring the resulting parallelism.

The metric used to measure the amount of parallelism of each application is *abstract speedup*. The abstract speedup is defined as the ratio of the number of cycles that the sequential execution of an application takes to the number of cycles that the parallel execution of an application takes to execute on the abstract system with multiple processors. Since an upper bound of performance is measured, the overhead of identifying and scheduling traces is not considered.

For each application, three sets of experiments are performed. In the first set, the amount of parallelism available in the benchmarks is measured for different numbers of processors. The results demonstrate the amount of parallelism available and how well the parallelism scales for the different benchmarks. In the second set, parallelism is measured while varying the communication type and cost. The results demonstrate the sensitivity of the parallelism to the cost of handling dependences at runtime. In the third set, parallelism is measured while varying the effect of the trace window. The results show the extent to which trace execution needs to be predicted accurately for parallelism to exist. Furthermore, reads are classified and measured based on where they obtain their data to better understand the factors that affect performance.

6.1. Effect of the processor count

Fig. 4 shows the abstract speedup of the benchmarks when the default system contains two, four, and eight processors. The benchmarks are listed according to benchmark suite. The first five benchmarks on the left are from Java Grande, the next six are from SPECjvm98, and the last ten are from Jolden. The speedups between benchmarks in the same suite, and therefore of relatively similar type, vary considerably. Thus benchmark performance cannot be easily classified according to type. The values of the different parameters shown in the figure represent a good trade off between overly pessimistic and optimistic assumptions. The effects these parameters have on performance are examined in subsequent sections.

Six applications, *moldyn*, *raytracer*, *euler*, *jess*, *em3d*, and *health*, exhibit speedups of more than four when the system has eight processors. These applications contain a large amount of trace-level parallelism, which may be exploited on systems with eight or more processors. Seven other applications, *search*, *db*, *mpegaudio*, *jack*, *bh*, *perimeter*, and *voronoi* exhibit speedups of more than two on both four and eight processors. These applications contain a considerable amount of parallelism. However, the amount of parallelism does not scale to a larger number of processors. The remaining benchmarks have very little parallelism available. Thus, the results indicate that the majority of benchmarks have enough trace-level parallelism to potentially take advantage of up to four processors, and some benchmarks can potentially take advantage of more than four processors.

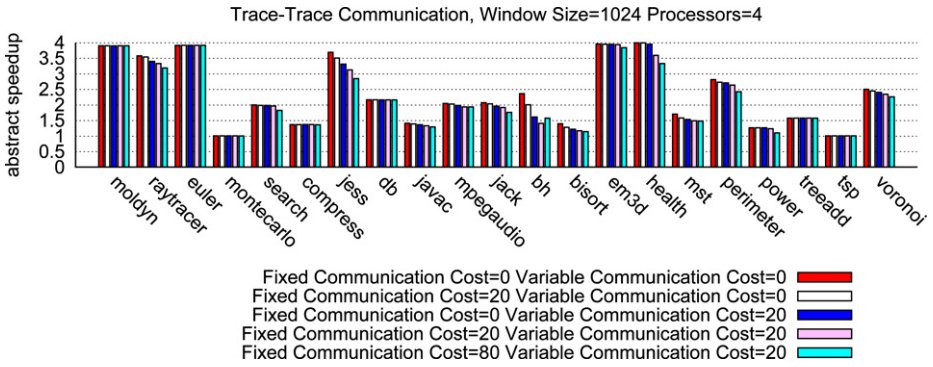

Fig. 5. Effect of the communication cost.

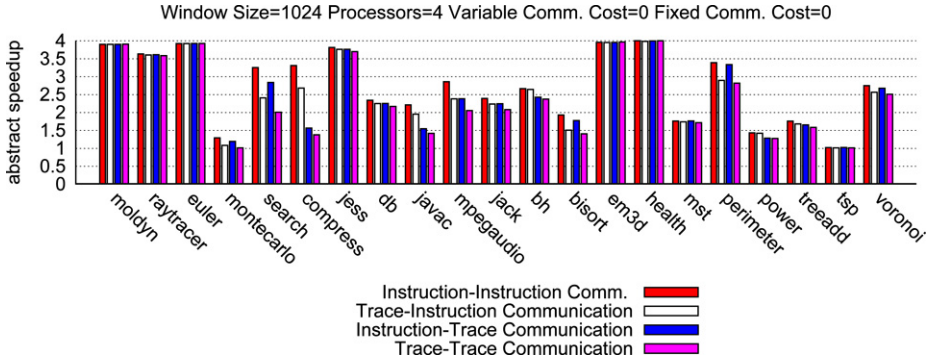

Fig. 6. Effect of the communication type.

6.2. Effect of the communication model

Fig. 5 contains the abstract speedup of the benchmarks when the fixed and variable communication costs are varied. Five configurations are shown. The first four have a combination of fixed and variable communication costs of 0 or 20 cycles. The last one has a fixed cost of 80 cycles and a variable cost of 20 cycles.

Three of the Grande and SPECjvm98 benchmarks, *raytracer*, *jess*, and *jack*, show sensitivity to the communication cost. The other benchmarks in these two suites show very little sensitivity. Therefore, the potential parallelism may be achieved even on systems that have considerable communication costs between processors. In contrast, approximately half of the Jolden benchmarks show sensitivity to the communication cost: *bh*, *bisort*, *health*, *perimeter*, and *voronoi*. Thus, parallelism may be hindered by systems that have large communication costs.

Fig. 6 contains the abstract speedup of the benchmarks when the communication type is varied. The effects of trace–trace (TT), trace–instruction (TI), instruction–trace (IT), and instruction–instruction (II) communication are evaluated.

The results indicate that the amount of parallelism is affected significantly by the type of communication used. Only seven benchmarks, *moldyn*, *raytracer*, *euler*, *em3d*, *health*, *mst*, and *tsp*, show no effect. Thus, being able to communicate dependence information between instructions instead of between the beginnings and ends of traces is important to exploiting the maximum amount of parallelism of the applications.

Furthermore, the results also indicate that the performance of instruction–trace communication is between that of trace–trace and that of instruction–instruction communication, and similar to trace–instruction communication.

Figs. 5 and 6 also indicate that the default configuration, which uses trace–trace communication with a fixed cost of 20 and a variable cost of 0, avoids excessive speedups based on extremely fast communication and presents a good middle ground for the configurations that use trace–trace communication. Thus, although it is not clear which configuration of the abstract system is the most realistic one, the default configuration is a good candidate to be analyzed in more detail.

6.3. Effect of the scheduling window

Fig. 7 depicts the abstract speedup of the benchmarks for four processors when the trace window size is 128 traces, 1024 traces, and 8192 traces.

For most applications, the window size has a small influence on the amount of parallelism available. However, there are several exceptions: *mpegaudio*, *health*, *perimeter*, and *voronoi*. For these benchmarks, a large number of traces will need to be scheduled together and the control flow between them will need to be predicted accurately. Thus, in these applications,

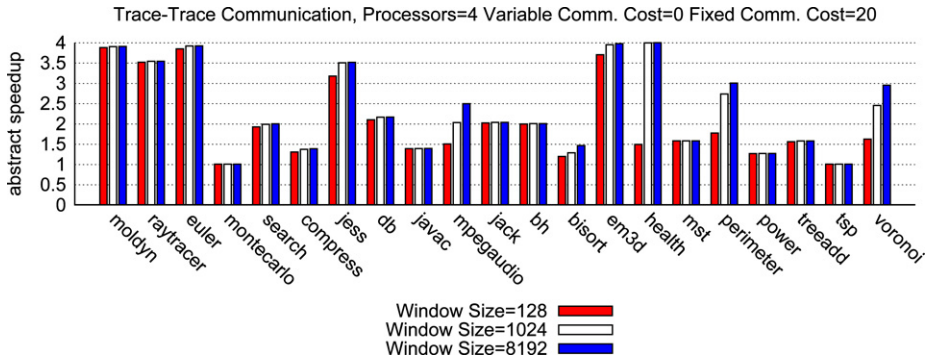

Fig. 7. Effect of the trace window size.

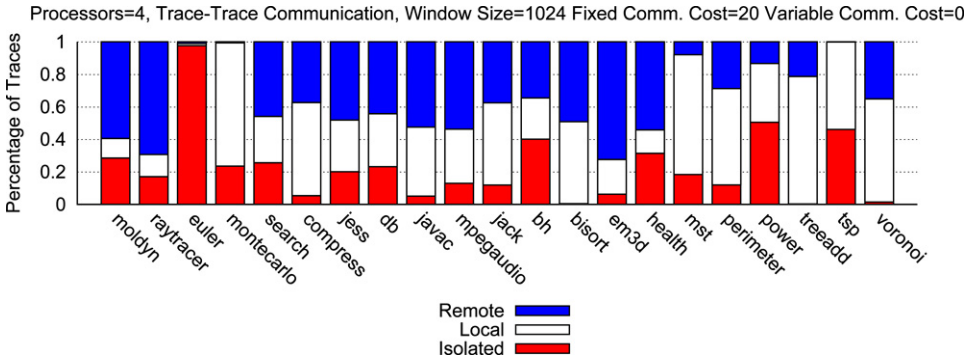

Fig. 8. Distribution of traces.

having a larger window improves parallel performance, but requires better prediction of trace execution. As discussed in Section 4, such prediction is inaccurate when the window size becomes larger and there are more traces for which execution needs to be predicted. Therefore an automatic parallelization system may have difficulty reaching the potential level of parallelism available.

Furthermore, although there is no speedup for *montecarlo*, a careful examination of the benchmark shows that it has parallelism at a very coarse granularity. Enabling the application to run in parallel would require an unreasonably large trace window. Thus, for that benchmark, a more complex analysis would be required to capture the inherent parallelism. Similarly, the Jolden benchmarks are based on algorithms that have very coarse grain parallelism, and therefore would also require a more complex analysis to capture inherent parallelism. Nonetheless, even without such an analysis, half of the Jolden benchmarks show good speedups.

Therefore it is possible to conclude that for most benchmarks, the instruction window can be small, and therefore control flow prediction does not have to be extremely accurate over a large number of traces for parallelism to exist.

6.4. Communication among traces

In this section, application performance is examined in more detail. Since our processor model is rather abstract, dependence among traces is the main factor that limits parallelism. Therefore, traces and their data accesses are characterized according to the classification in Section 4.3. First, a number of distributions is presented: the distribution of the different types of traces, the distribution of reads on these traces, and the distribution of reads on remote traces for the default configuration. Then the same graphs are shown for a system with more processors and a larger window size to show that the patterns hold for multiple abstract systems.

Fig. 8 presents the distribution of isolated, local, and remote traces. The *y*-axis contains the percentage of the number of traces in each group relative to all traces in each benchmark. Benchmarks with few local traces tend to perform well. In particular, out of the six benchmarks identified in Section 6.1 as having a large amount of parallel execution, five benchmarks, *euler*, *moldyn*, *raytracer*, *em3d*, and *health*, have the fewest local traces, 20% or less, relative to all traces. Also, these six benchmarks identified in Section 6.1 fall into two categories. One of the benchmarks, *euler* has a large number of isolated traces without dependences on other traces and therefore the scheduler can easily schedule the traces. The other five benchmarks have many remote traces and therefore have many dependences. However, these dependences were satisfied when the traces were scheduled. Therefore they exhibit good speedups. Conversely, benchmarks that have a considerable portion of local traces (e.g. *mst* and *montecarlo*) all perform poorly.

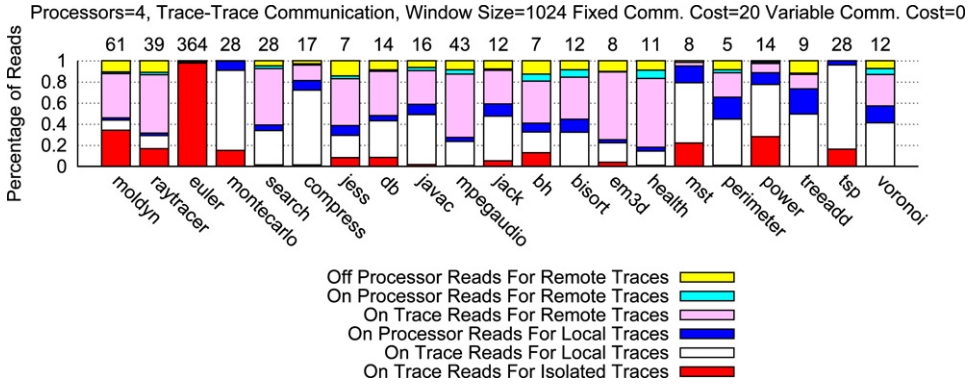

Fig. 9. Distribution of reads.

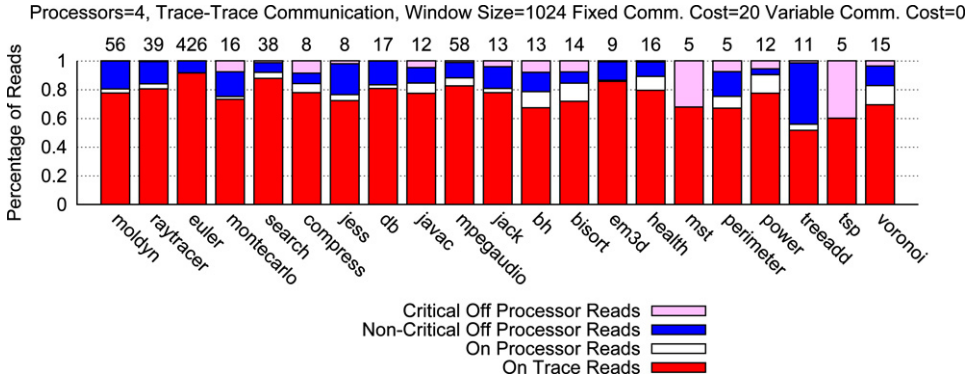

Fig. 10. Distribution of reads on remote traces.

Fig. 9 shows the distribution of on-trace, on-processor, and off-processor reads categorized further by which type of trace these reads are on: isolated, local, or remote. In total there are six different types of reads. The *y*-axis contains the percentage of each type of read relative to all reads. Furthermore, the number above each bar represents the average number of all reads per trace for that benchmark (these numbers are rounded to the nearest integer in all graphs). Again, these results are for the default system.

In Fig. 9, most benchmarks that do not exhibit good speedup also have more than 70% of all reads on local traces. This further supports the earlier observation that benchmarks that perform poorly do so because of lack of parallelism (i.e. a considerable number of local traces and reads) and not because of communication overhead. This figure also explains why some benchmarks tend to not be influenced by communication type, which is discussed in Section 6.2. Benchmarks are not influenced much by communication type if they have a small percentage of reads on local traces (less than 20%). These benchmarks are *moldyn*, *raytracer*, *euler*, *em3d*, and *health*. This behaviour implies that most reads are on isolated or remote traces, which are either independent or are dependent in a way that allows traces to execute in parallel in many different combinations. Therefore, the performance of these benchmarks is not affected by the type of communication, and the amount of parallelism exceeds the number of available processors. In contrast, benchmarks that have a large ratio of local reads lack sufficient parallelism. Thus, changes in the type of communication may increase the amount of parallelism, which can in turn be exploited by available processors. Thus, these benchmarks will tend to show sensitivity to the type of communication use.

Fig. 10 contains the distribution of the three different types of reads on remote traces. The figure further subdivides off-processor reads into critical and non-critical reads. The *y*-axis contains the four different types of reads and the number above each bar is the average number of reads per remote trace. Applications that have more critical off-processor reads than non-critical off-processor reads also have poor performance. Five benchmarks have this characteristic: *compress*, *bisort*, *mst*, *power*, and *tsp*. However, the performance of most of these benchmarks is also hindered by having too many local reads, indicating a lack of parallelism. Since these benchmarks have much fewer critical reads relative to local reads, lack of parallelism has a larger influence on performance.

The seven benchmarks that are hindered by too many critical reads or too many local reads have low speedups in Fig. 4. Furthermore, all benchmarks except for *javac* that are not hindered by these factors have good speedups. Therefore, whether or not an application has more than 70% local reads, shown in Fig. 9, and more critical than non-critical reads, shown in Fig. 9, is a good indication of whether speedup is present.

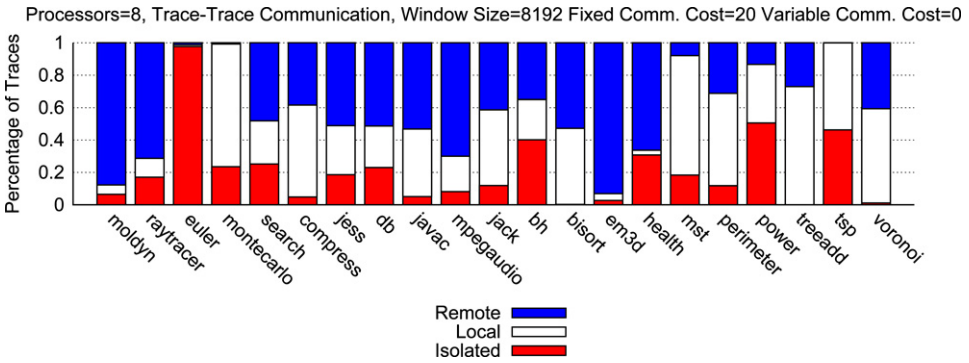

Fig. 11. Distribution of traces.

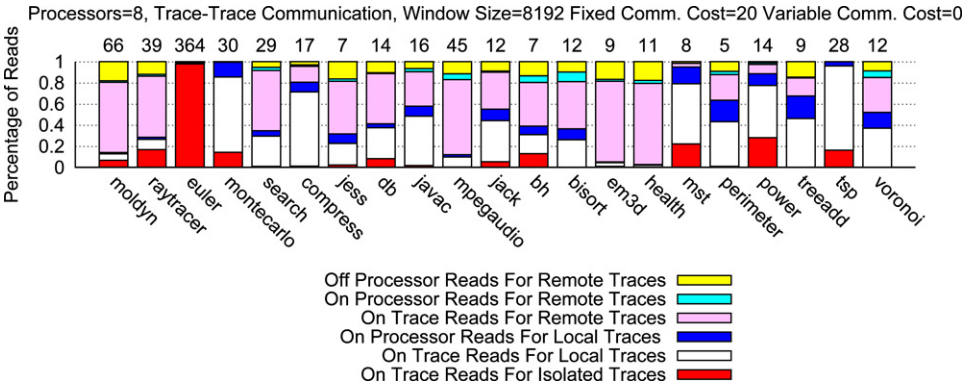

Fig. 12. Distribution of reads.

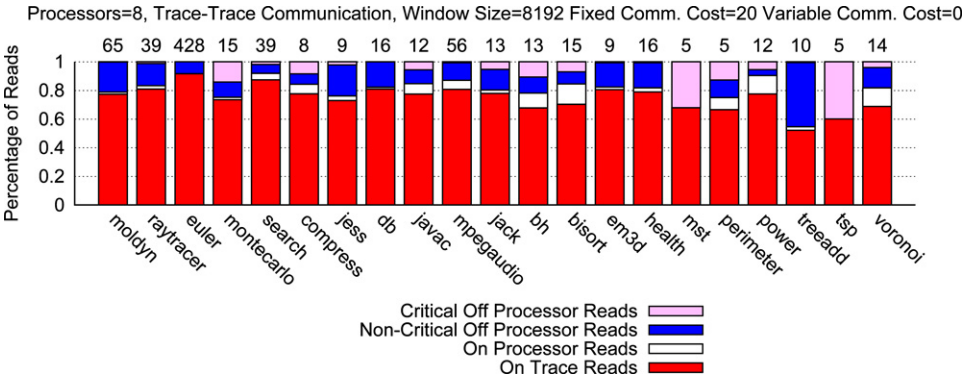

Fig. 13. Distribution of reads on remote traces.

The same statistics when the system being simulated has 8 processors, a fixed communication cost of 20 cycles, and a window of 8192 traces are in Figs. 11–13. The distributions of traces and reads are similar. This behaviour suggests that the underlying factors that affect performance are relatively independent of the number of processors and window size.

The underlying factors that affect performance are the lack of parallelism and communication overhead. Applications that have an abundance of parallelism (that is a low ratio of local traces/reads) tend to perform well even when critical reads are present, indicating that communication overhead can be hidden. Furthermore, for the minority of applications that do not have abundant parallelism in them, the number of local reads is significantly more than the number of critical reads. Thus, the impact of the local reads is more significant than the impact of critical reads on performance.

7. Related work

In this section, some of the research in automatic parallelization and using traces is briefly described and contrasted to our work.

A large amount of work has been done on traditional automatic parallelization [2]. Work has also been done on parallelization of loops at runtime, including work by Leung and Zahorjan [25] and Rauchwerger and Padua [27]. Other researchers, such as Chan and Abdelrahman [13], Johnson, Eigenmann, and Vijaykumar [24], and Obata, Ishizaka, and Kasahara [26] have focused on task-level parallelism. Speculation has also been examined by many, including Steffan et al. [30], Chen and Olukotun [14,15], and Du et al. [17]. The work on parallelization divides a program into partitions that are not related to traces. In contrast, we focus on traces as the unit of parallel work.

Traces have been used by researchers in traditional optimization. Fisher [18] uses traces for instruction scheduling. Ball and Larus [5] collect path statistics using path encodings. Ammons and Larus use path profiles to perform data flow analysis and constant propagation [1]. A number of runtime systems, such as HP Dynamo [3,4] and DynamoRIO [10,11], use traces to perform optimization at runtime. This work focuses on traditional optimizations. In contrast, we examine the potential of traces as units of parallel work.

Traces have also been used in hardware systems. Friendly, Patel, and Patt [19] use traces in the *fill unit* of a processor to perform optimizations. Jacobson, Rotenberg, and Smith [22] examine the predictability of traces in hardware and compare using traces to using hardware based multi-branch predictors. Furthermore, Rotenberg et al. [28] examine the effects of using a processor that has an architecture based on traces. This work focuses on using traces effectively on a single processor. In contrast, we examine the use of traces on multiple processors.

A number of researchers have also looked at the characterization of traces in Java programs. These include Bruening and Duesterwald [9], Berndl and Hendren [6], and Bradel [7]. These characterizations are all related to executing traces sequentially on a single processor. In contrast, we examine the execution of traces on multiple processors.

8. Conclusion

In this paper we have explored the availability of trace-level parallelism in Java programs. We have described a parallel execution model that uses traces to execute applications in parallel. We have presented an abstract parallel system that can execute traces in parallel. We have also described PIE, which simulates the execution of Java applications on this abstract parallel system. Based on the simulation results, we have found that sufficient parallelism exists for traces to be an effective unit of parallel work.

The experimental evaluation of 21 Java benchmarks indicates that most of the benchmarks have enough trace-level parallelism to be run effectively on systems with up to four processors. Furthermore, the trace window size has a small effect on the parallelism of most of the benchmarks. However, the amount of parallelism is affected significantly by the type of communication used. Therefore effective communication is essential to exploiting the full potential of trace-level parallelism. Observations made based on dependence characterization indicate that the main reason for reduced performance is a lack of parallelism. In contrast, communication cost is not a bottleneck because there are only a small number of critical reads. Overall, we conclude that enough trace-level parallelism exists for trace-based parallelization to be a potentially viable approach to improve the performance of Java programs.

Our model of the abstract parallel system is relatively simplistic. Although it does not provide realistic performance numbers, we believe that it provides a reasonably accurate indication of the amount of parallelism that exists among traces. Therefore, we believe that making the simulator more realistic will not change the qualitative conclusions.

Our future work is directed towards the implementation of an infrastructure that performs automatic trace-based parallelization efficiently on existing hardware. We hope that the infrastructure will be used either as an online or an offline feedback directed system and that the infrastructure will be able to parallelize a wide variety of applications effectively through the use of a combination of compiler analysis and speculation support.

Appendix

A.1. Simulation methodology and timing

In this section, details of the simulation of the benchmarks are provided. More specifically, the number of bytecodes executed, the overall simulation time, and how the time is divided is presented.

Jupiter is executed four times for each benchmark. The first three times are for simulation. Each execution corresponds to simulating a different window size. The fourth time is for measuring the time spent by Jupiter and RedSpot without using PIE. These runs are referred to as "Window Size = 128", "Window Size = 1024", "Window Size = 8192", and "No PIE" respectively.

For each window size, 24 configurations are simulated by calling MCP multiple times. The configurations have combinations of either 2, 4, or 8 processors and eight communication costs and types. The eight costs and types are instruction–instruction, trace–instruction, instruction–trace, and trace–trace communication with fixed and variable costs of 0 cycles and trace–trace communication with 20 fixed and 0 variable cost, 0 fixed and 20 variable cost, 20 fixed and 20 variable cost, and 80 fixed and 20 variable cost. Twenty four configurations are simulated at once to amortize the overhead of collecting all the trace and dependence information over a large number of configurations.

The execution time is divided into four components. The time spent by Jupiter and Redspot, the time spent saving data for PIE to use, the time spent by PIE, and the time spent by MCP. These four different parts of simulation are referred to as *Jup/RS*, *State Save*, *PIE*, and *MCP*.

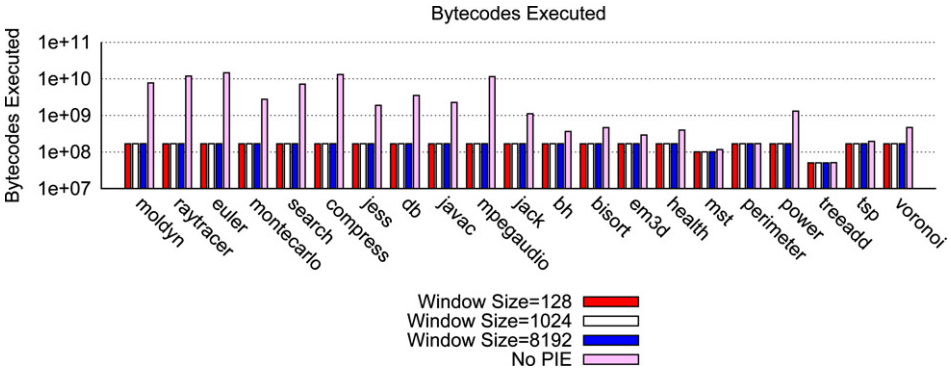

Fig. 14. Bytecodes executed.

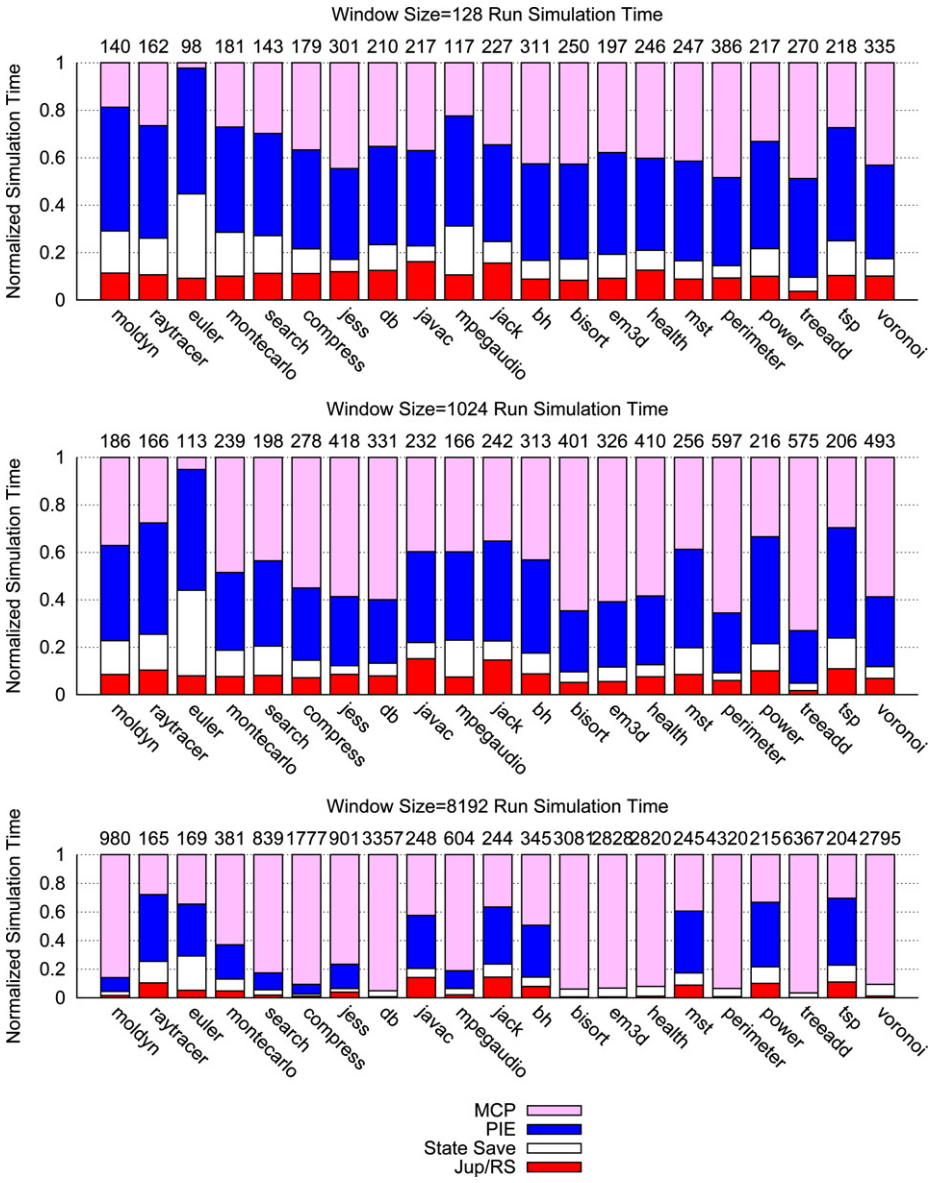

Fig. 15. Simulation time.

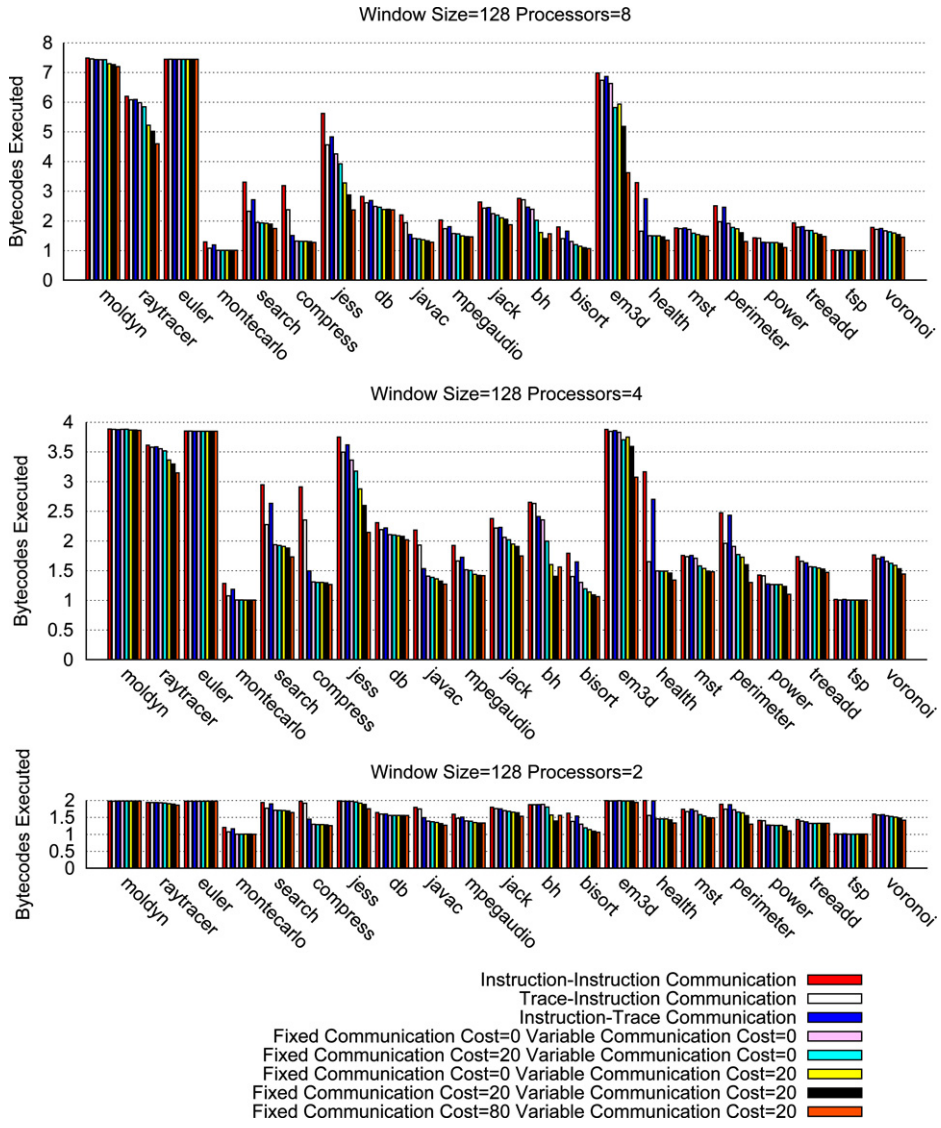

Fig. 16. Effect of the communication type for a window of size 128.

Jup/RS is calculated as the time that Jupiter and RedSpot would take to execute the number of instructions executed when not using PIE. It is the amount of time taken without PIE times the number of bytecodes executed with PIE and divided by the number of bytecodes executed without PIE. This calculation is an estimate of what the actual Jup/RS time should be based on the assumption that the average time per instruction between executions is the same. State save is the amount of time taken by Jupiter and RedSpot when using PIE minus the estimate of time taken without using PIE. This difference is the amount of overhead incurred by Jupiter and RedSpot in saving information for PIE. PIE is the amount of time for PIE to execute. And MCP is the amount of time for MCP to perform its scheduling of 24 different configurations.

Fig. 14 contains the number of bytecodes that are executed for each benchmark for the five runs. The *y*-axis is logarithmic. The number of bytecodes executed is shown since it is used to distinguish between the execution of Jupiter and RedSpot and the saving of state.

Fig. 15 contains the simulation time for the Window Size = 128, Window Size = 1024, and Window Size = 8192 runs. The *y*-axis for all three graphs is the percentage of time spent in each of the different parts of simulation relative to the total simulation time. The number at the top of each benchmark is the number of seconds of wallclock time that each execution took.

For the 128 and 1024 window sizes, both MCP and PIE each take approximately 40% of the time, while MCP dominates the simulation time when the window size is 8192. The reason for this behaviour is that MCP's execution time is proportional to the window size while the execution time of the other components is independent of window size.

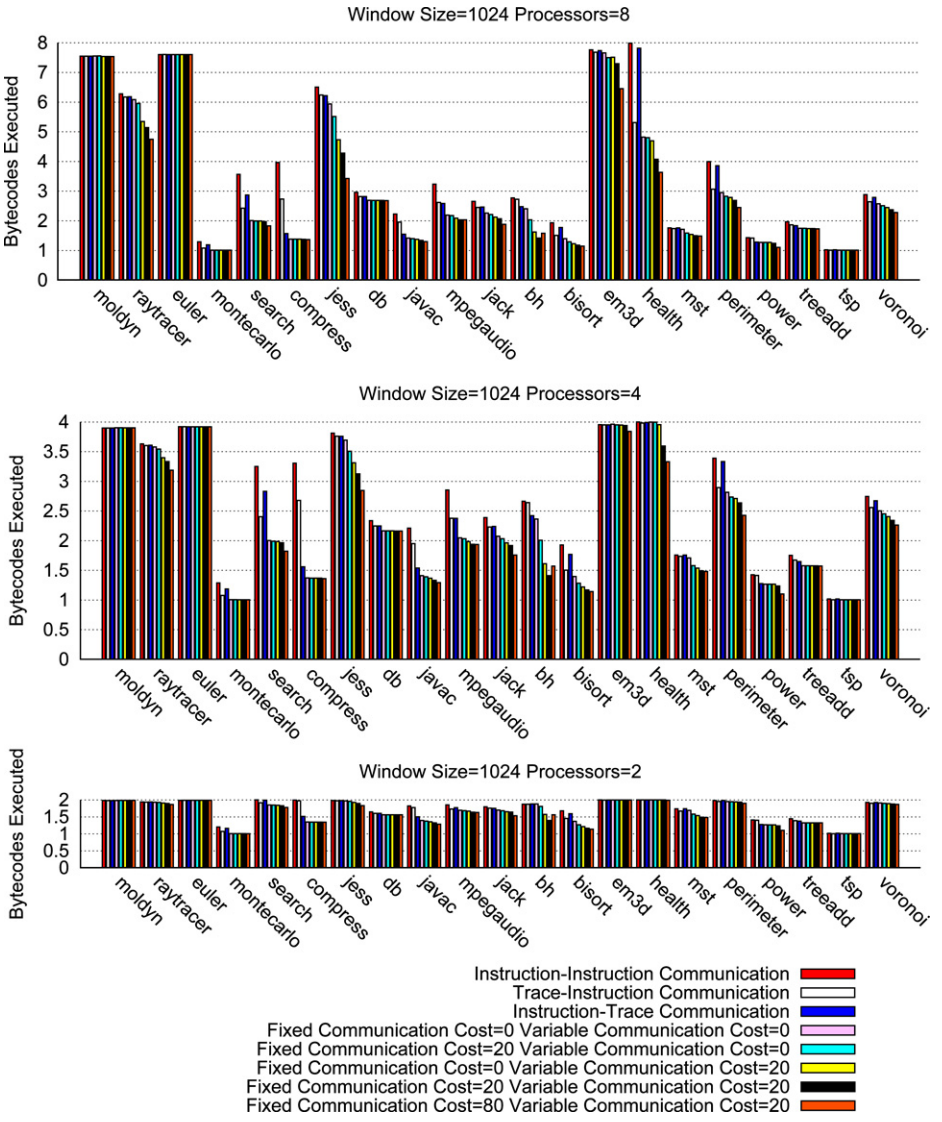

Fig. 17. Effect of the communication type for a window of size 1024.

If n is the total number of traces executed and w is the window size, then MCP is called $O(n/w)$ times and each invocation takes $O(w^2)$ time.[8] Therefore the total time complexity is $O(nw)$. The work of the other components is based on the total number of traces processed, which is $O(n)$.

The importance of amortization is also visible in these graphs. With amortization a single execution of Jupiter with RedSpot and PIE results in twenty four different configurations being simulated. Without amortization the non MCP parts of simulation would take twenty four times more time and would cause simulation time to increase significantly. Instead, with amortization, simulation time is relatively balanced between the different parts and tends to be dominated by MCP for the larger window size.

A.2. Complete performance results

For completeness, the graphs that show the effects of communication type and cost for all combinations of 2, 4, or 8 processors and 128, 1024, or 8192 trace window sizes are presented. Fig. 16 contains the graphs for a window of size 128, Fig. 17 contains the graphs for a window of size 1024, and Fig. 18 contains the graphs for a window of size 8192. These figures show that the data is robust across a variety of configurations and therefore add support to our conclusions.

[8] According to published work [32] the complexity for each invocation of MCP is $O(w^2 \log(w))$ while an online draft of a more recent paper [31] states that the complexity is $O(w^2)$. The latter complexity is used here.

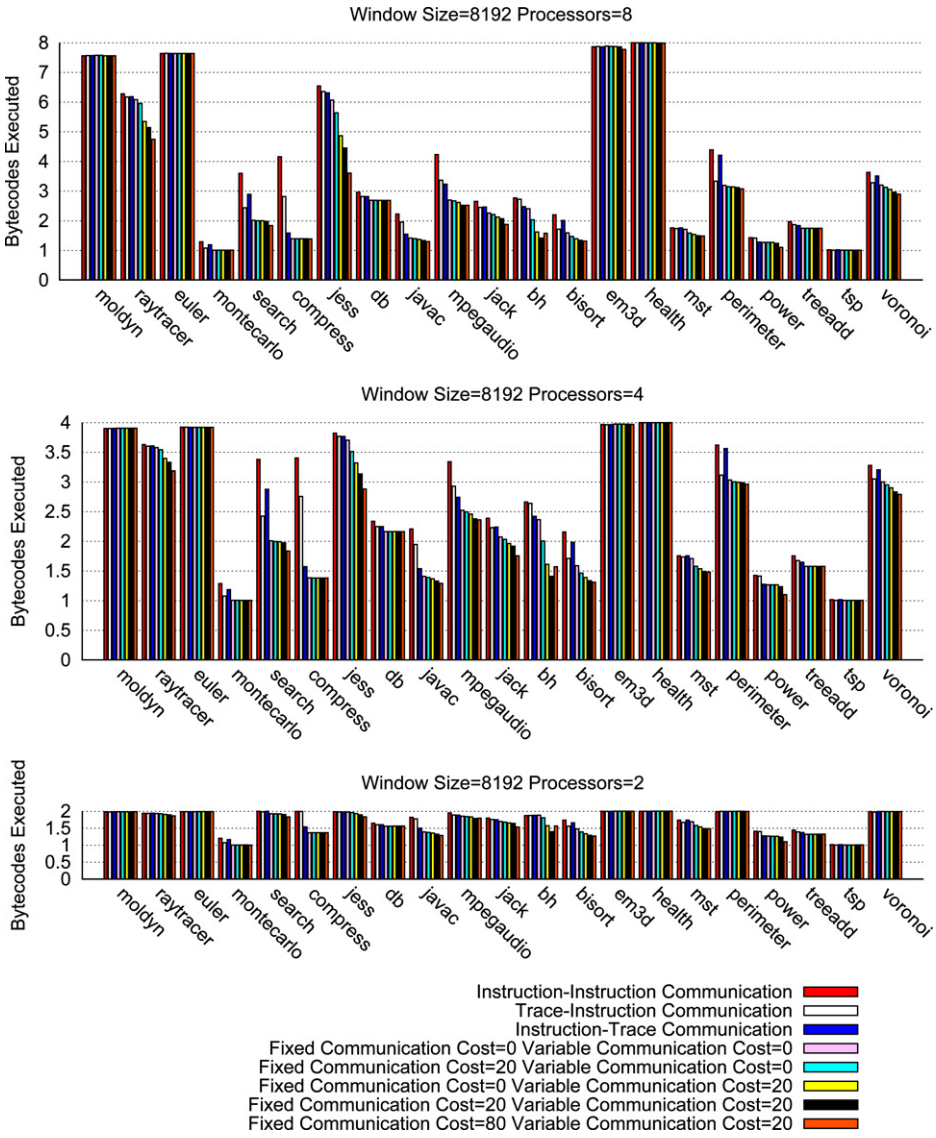

Fig. 18. Effect of the communication type for a window of size 8192.

References

[1] G. Ammons, J.R. Larus, Improving data-flow analysis with path profiles, in: Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, 1998, pp. 72–84.
[2] D.F. Bacon, S.L. Graham, O.J. Sharp, Compiler transformations for high-performance computing, ACM Computing Surveys 26 (4) (1994) 345–420.
[3] V. Bala, E. Duesterwald, S. Banerjia, Transparent dynamic optimization: The design and implementation of dynamo, HP Labs Technical Report, 1999.
[4] V. Bala, E. Duesterwald, S. Banerjia, Dynamo: A transparent dynamic optimization system, ACM SIGPLAN Notices 35 (5) (2000) 1–12.
[5] T. Ball, J.R. Larus, Efficient path profiling, in: Proc. of the Microprogramming Workshop, Micro, 1996, pp. 46–57.
[6] M. Berndl, L. Hendren, Dynamic profiling and trace cache generation, in: Proc. of the Int'l Symposium on Code Generation and Optimization, CGO, 2003, pp. 276–285.
[7] B.J. Bradel, The use of traces in optimization, Master's Thesis, University of Toronto, 2004.
[8] B.J. Bradel, T.S. Abdelrahman, Automatic trace-based parallelization of Java programs, in: Proc. of the Int'l Conference on Parallel Processing, ICPP, Xi'an, ISBN: 978-0-7695-2933-2, 2007, 26 pp.
[9] D. Bruening, E. Duesterwald, Exploring optimal compilation unit shapes for an embedded just-in-time compiler, in: Proc. of the 3rd Workshop on Feedback-Directed and Dynamic Optimization, FDDO, 2000.
[10] D. Bruening, E. Duesterwald, S. Amarasinghe, Design and implementation of a dynamic optimization framework for windows, in: Proc. of the 4th Workshop on Feedback-Directed and Dynamic Optimization, FDDO, 2001.
[11] D. Bruening, T. Garnett, S. Amarasinghe, An infrastructure for adaptive dynamic optimization, in: Proc. of the Int'l Symposium on Code Generation and Optimization, CGO, 2003, pp. 265–275.
[12] B. Cahoon, K.S. McKinley, Data flow analysis for software prefetching linked data structures in Java, in: Proc of the Int'l Conference on Parallel Architectures and Compilation Techniques, PACT, 2001, pp. 280–291.
[13] B. Chan, T.S. Abdelrahman, Run-time support for the automatic parallelization of Java programs, in: Proc. of the Int'l Conference on Parallel and Distributed Computing, PDCS, 2001, pp. 113–120.

[14] M. Chen, K. Olukotun, The Jrpm system for dynamically parallelizing Java programs, in: Proc. of the Int'l Symposium on Computer Architecture, ISCA, 2003, pp. 434–446.
[15] M. Chen, K. Olukotun, Test: A tracer for extracting speculative threads, in: Proc. of the Int'l Symposium on Code Generation and Optimization, CGO, 2003, pp. 301–312.
[16] P. Doyle, Jupiter: A modular and extensible Java virtual machine framework, Master's Thesis, University of Toronto, 2002.
[17] Z. Du, et al., A cost-driven compilation framework for speculative parallelization of sequential programs, in: Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, 2004, pp. 59–69.
[18] J.A. Fisher, Trace scheduling: A technique for global microcode compaction, IEEE Transactions on Computers C-30 (7) (1981) 478–490.
[19] D.H. Friendly, S.J. Patel, Y.N. Patt, Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors, in: Proc. of the Microprogramming Workshop, Micro, 1998, pp. 173–181.
[20] D. Geer, Chip makers turn to multicore processors, IEEE Computer 38 (5) (2005) 11–13.
[21] Jolden, University of massachusetts amherst architecture and language implementation laboratory, 2003. ftp://ftp.cs.umass.edu/pub/osl/benchmarks/jolden.tar.gz.
[22] Q. Jacobson, E. Rotenberg, J.E. Smith, Path-based next trace prediction, in: Proc. of the Microprogramming Workshop, Micro, 1997, pp. 14–23.
[23] Java grande forum benchmark suite, 2003. http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html.
[24] T. Johnson, R. Eigenmann, T.N. Vijaykumar, Min-cut program decomposition for thread-level speculation, in: Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, 2004, pp. 59–69.
[25] S. Leung, J. Zahorjan, Improving the performance of runtime parallelization, in: Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP, 1993, pp. 83–91.
[26] M. Obata, K. Ishizaka, H. Kasahara, Automatic coarse grain task parallel processing using OSCAR multigrain parallelizing compiler, in: Proc. of the Int'l Workshop on Compilers for Parallel Computers, CPC, 2001, pp. 173–182.
[27] L. Rauchwerger, D. Padua, The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization, in: Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, 1995, pp. 218–232.
[28] E. Rotenberg, et al., Trace processors, in: Proc. of the Microprogramming Workshop, Micro, 1997, pp. 138–148.
[29] SPEC, Standard performance evaluation corporation, 2003. http://www.specbench.org/.
[30] J.G. Steffan, et al., A scalable approach to thread-level speculation, in: Proc. of the Int'l Symposium on Computer Architecture, ISCA, 2000, pp. 1–24.
[31] M.-Y. Wu, MCP source code, University of New Mexico, 2007. http://www.eece.unm.edu/~wu/mcp/.
[32] M.-Y. Wu, D. Gajski, Hypertool: A programming aid for message-passing systems, IEEE Transactions on Parallel and Distributed Systems (TPDS) 1 (3) (1990) 330–343.
[33] T. Yasue, et al., Structural path profiling: An efficient online path profiling framework for just-in-time compilers, The Journal of Instruction Level Parallelism 6 (2004) 1–28.